Making R Graphs, For People Who Don't Want To Learn R

Richard Blissett

I get it. You already know one statistical programming language, and the idea of having to learn another one just to make pretty pictures doesn't seem worth it to you. I'd argue that there are other benefits of learning R and that you're weak, but then I remembered that I thought about going for a run this morning but then changed my mind when I realized that I had leftover pizza in my fridge.

With that humility in mind, I present this guide as an opportunity for users of other statistical languages (Stata is discussed primarily here) to take advantage of R's arguably superior graphing abilities. I make no claim here about whether R or Stata is easier, since that depends on you, but I think it is more commonly agreed that one of R's strengths over other statistical programming languages is the control over and quality of the graphics it can produce, not to mention that the graphs tend to look much cleaner.[1] While yes, you will need to actually use the R environment, there is no need to learn the full syntax for the R language or even most common commands. Really, you will only need to know a handful to be successful.

The other critical skill you will need, which I'm sure you already have, is how to look things up on the internet. There are hundreds of examples of the kinds of graphs you can make in R, along with bits of code that you can copy and modify for your own purposes.

There are really just four steps that will be involved in creating these graphs:

1. Clean a set of your data in the statistical language of your choice to match the input format required of the particular graph you want to create.
2. Import that data into R.
3. Craft the command(s) you will use to create the graph.
4. Output the data to a file.

The main package and set of commands that we will be using here is the `ggplot2` suite. Developed by Hadley Wickham in 2005, this package has become one of the main tools for creating many of the graphics that you see created in R.

My goal here is to provide you with a base set of skills, akin to learning the base set of chords on guitar. You're not exactly a "guitar player," but you know enough to recognize that Jason Mraz's "I'm Yours" can be replicated with four basic chords and even enough to lay down something that could pass for really catchy at a tiki bar, should the need arise. My goal is to make you "that guy" (or girl, or wherever on the gender spectrum you find yourself), but unlike at the tiki bar, being "that guy" in the statistical graphics department is a good thing. Good luck.

[1] It is also entertaining to note that R graphics also have the wonderful quality that they can produce nonsense that is beautiful, if you mess something up. For a gallery of these types of mistakes, see http://accidental-art.tumblr.com/

Contents

1 Setting Up the R Environment **3**
- 1.1 Installing R . . . 3
- 1.2 The RStudio Environment . . . 3

2 Making R Graphs 101 **4**
- 2.1 Clean the data to specifications . . . 4
  - 2.1.1 Single-group graphs . . . 4
  - 2.1.2 Multiple-group graphs . . . 5
- 2.2 Import that data into R . . . 5
- 2.3 Craft the command(s) you will use to create the graph . . . 6
  - 2.3.1 A long example . . . 7
  - 2.3.2 Scatter plot . . . 11
  - 2.3.3 Box-and-whisker plot . . . 11
  - 2.3.4 Histogram . . . 12
  - 2.3.5 Bar plot . . . 12
  - 2.3.6 Conditional bar plot . . . 13
  - 2.3.7 Line plot . . . 15
  - 2.3.8 Re-using graph formats . . . 15
- 2.4 Output the data to a file . . . 16

1 Setting Up the R Environment

In this section, I'm going to tell you how to set up R on your computer and then run through a basic overview of what you're looking at, so it doesn't look like organic chemistry.[2]

1.1 Installing R

Do this list of things:

1. From the following website, click "CRAN," pick a mirror site (just pick the one nearest to you), select your operating system, and download/install the most recent version of R.
   Website: http://www.r-project.org/
2. From the following website, click "Download RStudio," choose "Desktop," and then download and install RStudio.
   Website: http://www.rstudio.com/
3. Open RStudio.
4. Get a glass a water, because hydration is important.
5. Within RStudio, go to the "Tools" menu, then click "Install Packages..."
6. It should say "Install from: Repository (CRAN)." If it doesn't say that, make it say that.
7. Type "ggplot2" where it asks for the package name, make sure that the "install dependencies" option is checked, and click "Install."
8. Do the last three steps again, but for the "haven" package.[3]

If the websites have changed since I have written this, I'm sorry. I think you can figure it out, though.

1.2 The RStudio Environment

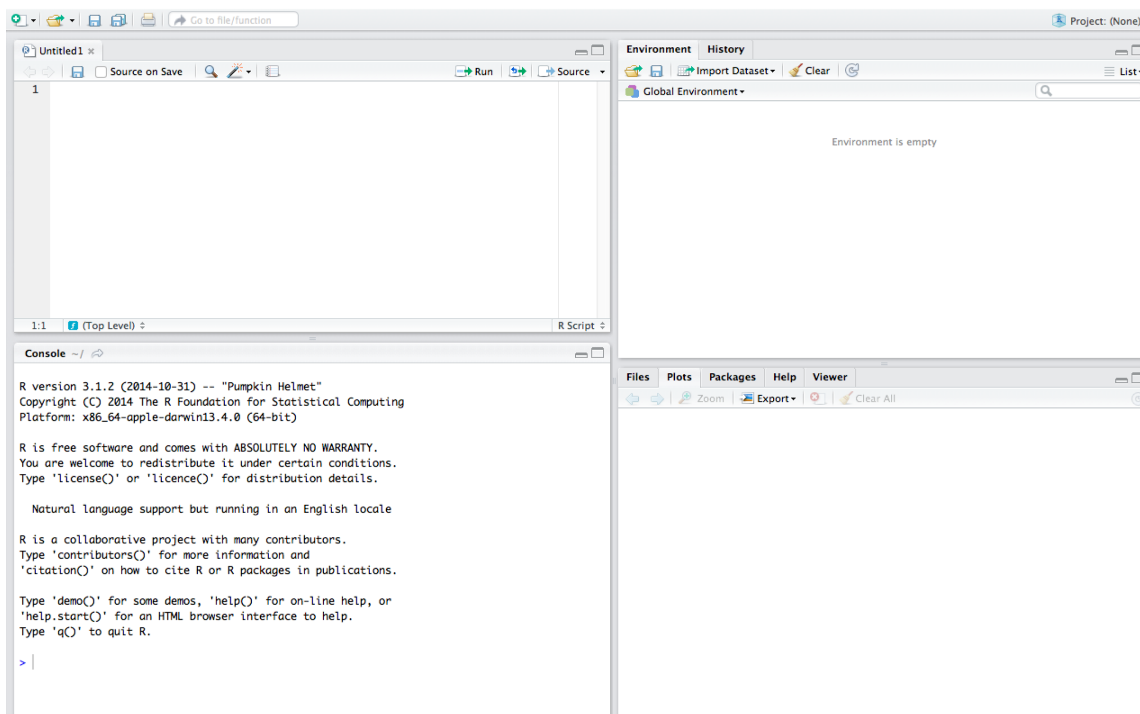
Here's the basic gist behind the two programs I just had you install: R is the language. It comes with it's own console and environment, like the Stata environment. However, it's a little clunky, and people tend not to like using it. As such, the wonderful people over at RStudio created an environment that uses your installation of the R language, but in a much better environment that is easier to use and understand.

So open RStudio. When you open RStudio, you should have four basic areas on your screen:

1. The top-left is where you type source code, like the script editor in Stata.
2. The bottom-left is the console, where you see what R is doing. Like in Stata, you can just type in the console next to the `>` character.
3. The upper-right is where you see you list of R objects and things in your workspace (not overly important for your purposes here). History is also there.
4. The bottom-right is where you can see your file structure, graphical output, available packages, and help documents.

It should look something like this.

[2] Unless, of course, organic chemistry looks perfectly comprehensible to you, in which case I'm sorry for your life choices.
[3] This step really only applies to Stata users. For the rest of you, ask an R friend how to import your data files into R.



If you don't have the top-left console, click "File," "New File," "R Script." You'll be working mostly in the top-left console, and you'll see the graphs in the bottom-right console.

To run any section of code, you will need to highlight the code and click "Run," which is at the top of the script console. Unlike Stata, you cannot just click "Run" to run your entire code. If you want to do that, you will need to select all of the text in your code, and then click "Run."

2 Making R Graphs 101

Here, I go through each of the four steps mentioned in the beginning of this document.

2.1 Clean the data to specifications

This is the main time saver for you. Rather than having to figure it all out in R, it's probably going to be most efficient for you to clean a set of the data, with only the variables you need, all the way up to what the R code needs in order to create your graphs.

2.1.1 Single-group graphs

For simple, single-group plots like scatter plots between two variables, simple density plots, histograms, and the like, the setup of the data is pretty straightforward and identical to what you would need in Stata.

1. Scatter plot: x values in one column, y values in another column
2. Bar plot: categorical x values in one column
3. Histogram: continuous x values in one column

And so on and so forth...

2.1.2 Multiple-group graphs

For graphs over multiple groups, typically, the `ggplot2` commands will want your data in *long* format. For example, let's say you want density plots of the number of hamburgers that people can eat in a sitting, but you wanted separate density plots for Catholics, Methodists, and Athiests. The logic of `ggplot2` will ask for data that is at the individual-by-religion level so that it can plot the individual level data, but determine to which plot the data is going based on the religion of the individual.

While this kind of setup, for subgroups, is probably straightforward, where this is also important, but probably less intuitive given how we tend to treat data, is how to, for example, plot the density plots for different variables (e.g., four different measures of teacher quality in order to compare distributions). Here, you will still need your data wide, where one column has the values, and the other column has the labels for the different values. (There are also ways to more simply plot different plots on top of each other, but that is beyond the scope of this document.)

2.2 Import that data into R

Finally, some actual R coding.

First, if you haven't already done so, open RStudio. Like we typically do in Stata, you'll want to first set your working directory. This is where your program will look for files if you don't provide it another path, and it is also where it will output any files that don't have another path. To do this, use the `setwd()` command, with the directory path in parentheses. Type the following into your R script (except with your own working directory, of course).

```
# Sets the working directory
setwd("~/Desktop/MakingRGraphs")
```

Second, type the next two lines. They load the libraries I had you install earlier.

```
# Loads libraries
library(ggplot2)
library(haven)
```

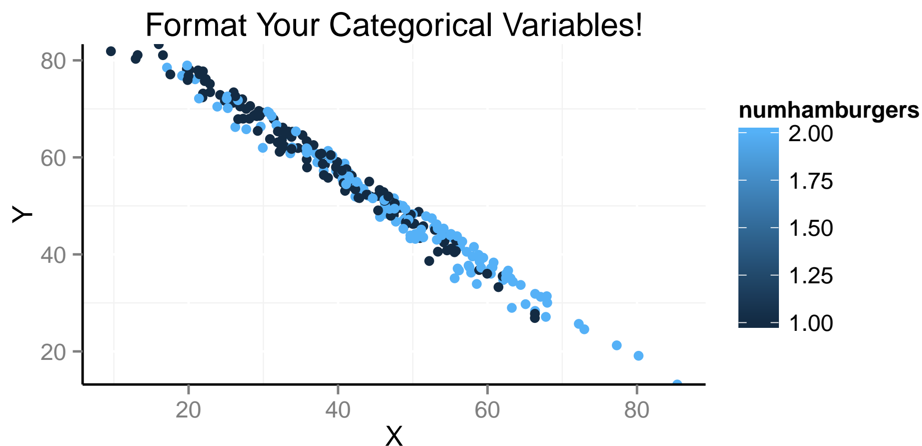
Finally, import your data. Make sure to add the ".dta" file extension.

```
# Reads in data
d <- read_dta("mydata.dta")
```

The data is now stored in a container called "d."[4]

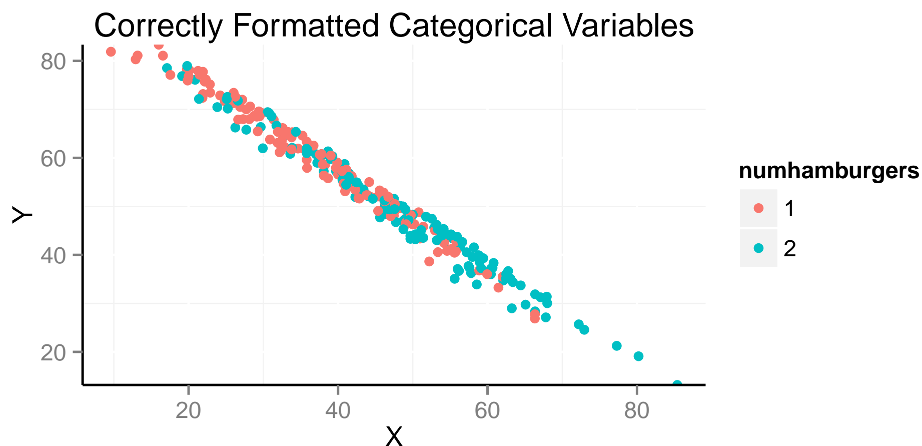
One important distinction between Stata and R is that categorical and numeric variables must be stored as separate formats. While string variables will automatically be stored as categorical, any all-number variables will be read into R as numeric, even if you want them to be categorical (e.g., the number of the month). See below, where I plot X against Y and ask for the color of the points to be based on a variable called "numhamburgers." This is what happens when you try to use a categorical variable that's formatted as numeric.

[4]What this line did was read your R data into an "object." Essentially, what this means is that whenever you bring in data, it doesn't stick it on your table and say, "Well, this is all we're working with now," like Stata and some other languages do. Instead, it sticks the data in a "container" (in R language, a "vector" or "object"), and you can have as many containers on the table as your poor computer's memory can handle. The upside to this logic, for R, is that you can have multiple data sources open at the same time. The necessary downside (which is really just a minor inconvenience) is that when referencing variables, we have to reference them by their container name as well as the variable name. So while, in Stata, we would just type `hamburgers` whenever we wanted hamburger information, in R, we have to type `d$hamburgers`. The reason for this is because you could also have an "f" container also with a "hamburgers" variable, in which case that data would be referenced with `f$hamburgers`, in order to distinguish between the hamburgers data in containers "d" and "f." You won't typically need to do that here, but in case you see that kind of notation elsewhere, this is what it is talking about.



Yes, R can color things as a gradient,[5] which looks nice, but that's not what we want. R thinks that "numhamburgers" is a continuous variable that could take on any value between 1 and 2. As you can see from the legend, then, it allows for all of those possible values. What you want, instead, is for R to treat "numhamburgers" as categorical. See the following code, where "d" is the container in which we put the data and "numhamburgers" is the variable that is numeric (with values of 1 and 2) but meant to be treated as categorical.

```
# Change number of hamburgers to be a categorical variable
d$numhamburgers <- factor(d$numhamburgers)
```



2.3 Craft the command(s) you will use to create the graph

This is admittedly the most complicated part of this, but once you get the basics (and know where to find good example code on the internet), it's all fairly easy.

The first thing to understand here is a little bit of logic. The `ggplot2` package is named as such because it is named after a concept called the "Grammar of Graphics," proposed by Leland Wilkinson, Anushka Anand, and Robert Grossman in 2005. We think of any human language as combinations of different parts, including nouns, verbs, and the like. All together, these parts create a whole language that conveys meaning

[5]Or, as dubbed by Sy Doan, "Fifty Shades of Blue."

and is a standardized way for people to understand each other. These authors proposed that we could think about scientific graphics the same way, with different data parts and parameters coming together to make a whole. This was modified by Hadley Wickham afterwards, where he proposed thinking of these parts as layers, but for our purposes, the distinction is not overly important.[6]

With that, this means that instead of thinking of a scatter plot, for example, as simply a graph that has different options you can modify, think of it as an axis, plus mapped points, plus a title, plus a legend (maybe), plus axis labels, plus an orientation...any aspect you can think of in a graph can be thought of separately. We will not have to specify each one of these parts, since for ease of use, there are defaults for each of them, but understanding graphs as combinations of parts is important for understanding `ggplot2`.

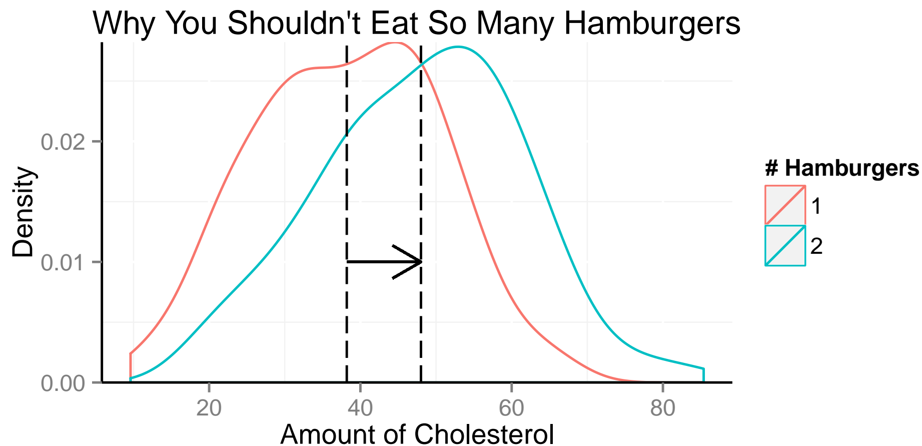
It is also important to note that also for ease of use, the `ggplot2` suite includes a `qplot()` function that wraps up some of the complicated stuff I'm about to show you into a single command for making simple, standard-format `ggplot2` graphs. Ironically, given the aim of this document, I'm not going to cover this command. This is for three reasons. First, I think keeping the "Grammar of Graphics" logic consistent throughout may be simpler in order to help people understand what is actually happening. Second, doing graphs without `qplot()` gives you a bit more control, and I'm all for power of the people. Lastly, I think it's fairly simple to look up the `qplot()` documentation, which is pretty clear, and learn `qplot()` yourself. Conversely, the `ggplot()` conventions, which we will use here, take a little getting used to, but are also your only options if you're going to do anything outside of the most standard graphs.

2.3.1 A long example

I think the easiest way to show this point and also start learning in the least pedagogically-sound way is to start with something more complicated, understand the point of how it's structured so it doesn't look so complicated anymore, and then move into simpler things. Below is some code to create a graph, and then the created graph below that.

```
ggplot(d) +
  geom_density(aes(x=cholesterol, colour=numhamburgers)) +
  geom_vline(xintercept=38.20, linetype="longdash") +
  geom_vline(xintercept=48.01, linetype="longdash") +
  geom_segment(aes(x=38.20, y=0.01, xend=48.01, yend=0.01),
               arrow=arrow(length=unit(0.5, "cm"))) +
  ggtitle("Why You Shouldn't Eat So Many Hamburgers") +
  xlab("Amount of Cholesterol") +
  ylab("Density") +
  scale_y_continuous(expand=c(0,0)) +
  scale_colour_discrete(name="# Hamburgers") +
  theme(panel.background=element_blank(),
        axis.line=element_line(colour="black"))
```

[6] To be fair, for our purposes, none of this history is overly important, but paying some respect to the wonderful nerds who had the patience to come up with this stuff is, I think, somewhat important in the grand scheme of things. Hats off.



This graph shows the distribution of the cholesterol levels for people who have eaten one hamburger, versus those who have eaten two (it might be good to see this in color).[7] The black dotted lines show the means of the "one hamburger" group versus the "two hamburger" group, and the arrow shows the shift between the two.

Looking at the R code, the basic structure is, as mentioned above, a series of parts, combined using the "+" symbol. Notice that each "part" is on a separate line. This is really just a stylistic choice that is totally up to you, though I'm going to go ahead and tell you that I have no desire to ever work with you and read your code.

What I'm going to do now is go through each part so we can understand what is happening. If you're just looking for a specific graph type, you can skip this example. However, I think it can help you think though how to make statistical art.

The language might seem like a lot of different words to remember, but here's the secret: I haven't memorized most of these lines either. I, as well as most of the normal world, just work from examples from my previous work or examples that I can find online. Pretty much all of what you will ever need to do with a graph, someone else has thought of and posted a solution online. And if you come up with something new, post it on Stack Exchange (usually the Stack Overflow community), and typically, you'll get an R buddy with an answer within a week or so. We try to help each other out here.[8]

```
ggplot(d)
```

This line is really the first, essential key. It tells R that you are now making a graph, and that the data stored in `d` is where it should look for the variables you are about to reference in the other lines.

```
geom_density(aes(x=cholesterol, colour=factor(numhamburgers)))
```

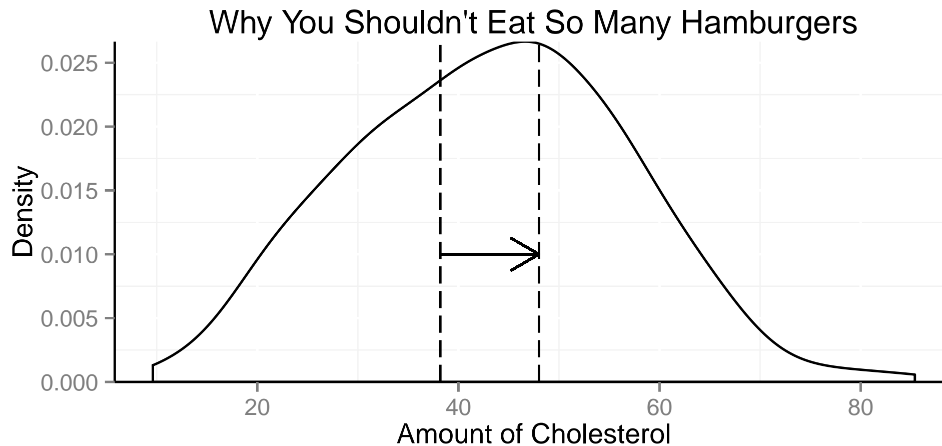
This is the second, essential key. The `geom_density()` function is a part of the `ggplot2` package, and it is used for adding density plots to a graph. There are a variety of related functions, including `geom_point()`, `geom_bar()`, `geom_histogram()`, `geom_line()`, `geom_violin()`, and `geom_boxplot()`. All of these functions are for adding shapes with certain parameters, and all of them have the same basic setup: use the `aes()` section to set up the basics, and then add any other options than come along with the function.

The `aes()` (for "aesthetic") section is the part that you probably care about most in this graph - what are we actually graphing? In the case of a density plot, you need to tell it just the variable for which it

[7]Note that this data is fabricated and is not a statement on health or nutrition (which I assume are related things). I have no idea in what units cholesterol is typically measured. I endorse the eating of as many hamburgers as you can eat without feeling guilty about world hunger. Have fun chewing on that.

[8]Do make a really diligent effort to find an answer somewhere else online or on Stack Overflow before posting your own question. You're not going to make any friends by posting a question for which it was easy to find the answer if you just searched for more than three seconds.

is plotting the density with the "x" setting. If you have multiple categories over which would like to plot the data in different colors, you use the "colour" setting to specify that variable.[9] If you didn't include this option, it would just plot all of the data in one density plot, as seen below (though the dotted lines now have no meaning).



Alternatively, if you wanted to instead have the different groups differentiated by the line type, rather than color (for black and white graphs), you would just use a "linetype" option, rather than "colour."

You could really just make something kind of nice looking with these first two lines, but it would be a bit unprofessional looking, like you didn't really try. I know that's the underlying motivation behind this document, but really, have some standards.

```
geom_vline(xintercept=38.20, linetype="longdash")
```

This line and the next one add the vertical dotted lines, using the `geom_vline()` command. I calculated the means in Stata and hardcoded them here, which is the lazy way to do it, but there are ways to make this automated and calculated directly in R. I also specified that I wanted the line type to be "longdash," one in a series of different line types you can have in Stata. The default is a solid black line. Note that you could also specify the color here using the "colour" option. You can look online for the different color options available, and this option also accepts conventional hex codes for colors.[10] Go crazy.

```
geom_segment(aes(x=38.20, y=0.01, xend=48.01, yend=0.01),
             arrow=arrow(length=unit(0.5, "cm")))
```

This line (or lines, since I broke it up to avoid a super long line) makes the arrow. After all, an arrow is just a line segment with a pointer on one end. The short version of the story is that this function takes the beginning and end coordinates of the line, and the "arrow" option[11] tells it that it's an arrow and also how long the little pointer lines should be. (I'm sure there's some name for these lines. I don't know what it is.)

```
ggtitle("Why You Shouldn't Eat So Many Hamburgers")
```

This adds the title. If you don't add this, there's usually no title. Simple.

`xlab("Amount of Cholesterol")` and `ylab("Density")`

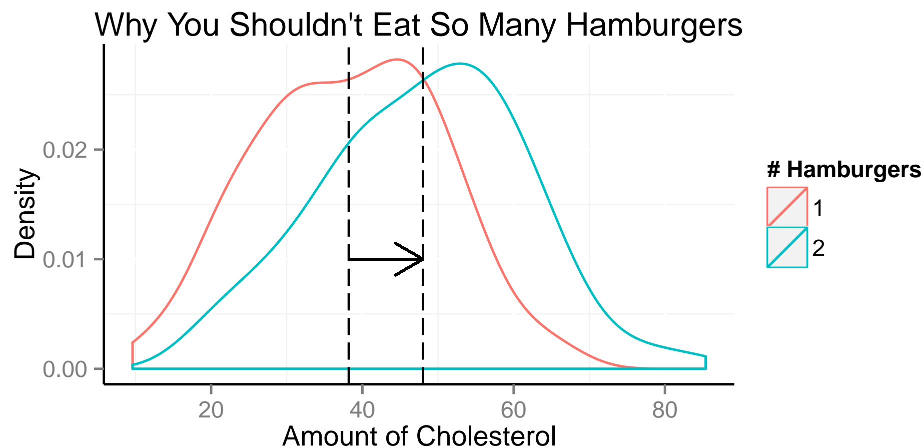
These lines add the labels for the x and y axes. Without these labels, the graph will just use the variable name or the default name of the metric it is using for the particular graph.

```
scale_y_continuous(expand=c(0,0)))
```

[9] Before any of you Americans get your knickers in a twist, Hadley Wickham is from New Zealand.
[10] You could have also included a specific color for a line in the aesthetic for the `geom_density()` example above.
[11] You'll need the `grid` library if you want to include an arrow.

The `ggplot()` suite has a quirky default that puts the zero point for axes a little bit off of the actual corner of the axes, as shown below.

```
ggplot(d) +
  geom_density(aes(x=cholesterol, colour=numhamburgers)) +
  geom_vline(xintercept=38.20, linetype="longdash") +
  geom_vline(xintercept=48.01, linetype="longdash") +
  geom_segment(aes(x=38.20, y=0.01, xend=48.01, yend=0.01),
               arrow=arrow(length=unit(0.5, "cm"))) +
  ggtitle("Why You Shouldn't Eat So Many Hamburgers") +
  xlab("Amount of Cholesterol") +
  ylab("Density") +
  scale_colour_discrete(name="# Hamburgers") +
  theme(panel.background=element_blank(),
        axis.line=element_line(colour="black"))
```



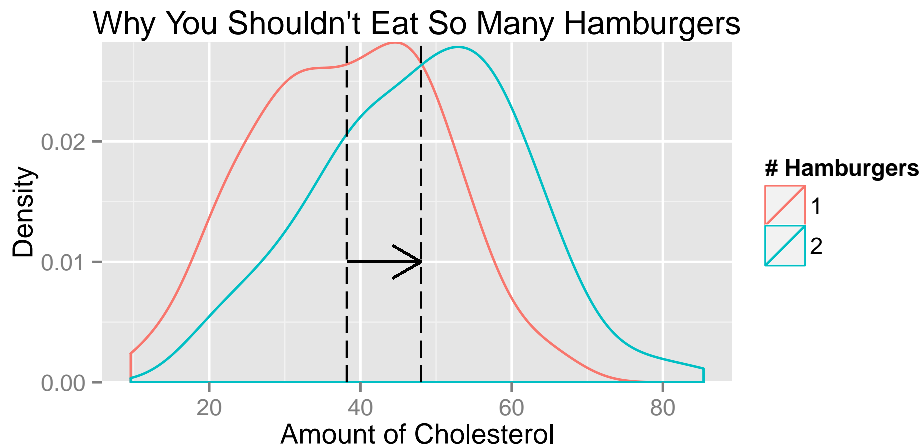
I suppose that looks good and classy in some contexts, but I didn't want that here, so I included the line above. Why did I include such a specific tweak for the purposes of learning? To show you that anything is possible. Shoot for the moon, my children, because if you miss, you'll probably end up drifting through space until you run out of oxygen and die. Or you might hit a star, which doesn't seem any better.

```
scale_colour_discrete(name="# Hamburgers")
```

I'm just telling R that the name of the color scale I am using is "# Hamburgers" and as such, that's what it should name the legend. Otherwise, it will name the legend the name of the variable, "numhamburgers."

```
theme(panel.background=element_blank(),
      axis.line=element_line(colour="black"))
```

The `theme()` function is basically the "Any other general theme things that you want to change about your graph" function. The default theme for `ggplot()` typically looks like the following.



It's nice for just sharing graphs with friends (if you have friends, in which case call me because I have questions). But it's not exactly publication-worthy, nor does it look that clean. As such, the `panel.background` option tells R to remove the background panel, and `axis.line` option adds the axis lines. This is also where you would assign colors of your axes and axes labels, change font and sizes, and other nitpicky things.

So that's all for this graph. Here are some common examples of graphs for your perusal. (Ignore any and all units. I don't know arbitrary units.)

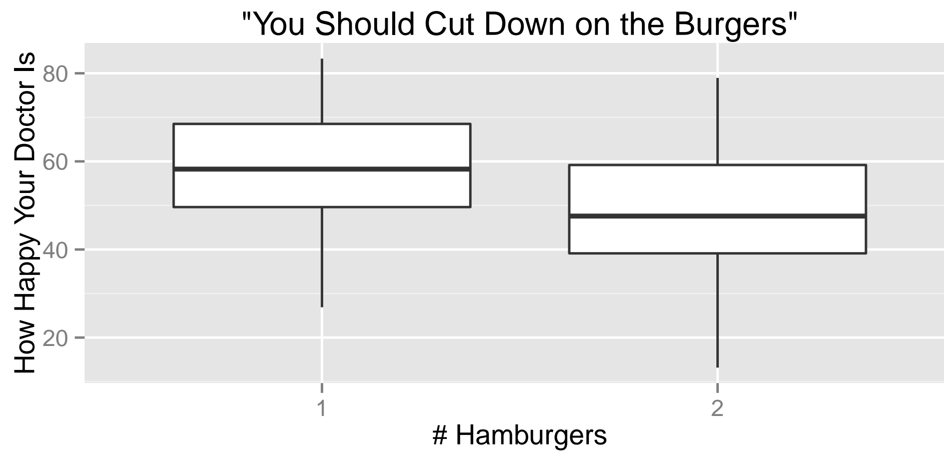
2.3.2 Scatter plot

```
ggplot(d) +
  geom_point(aes(x=cholesterol, y=dochappy)) +
  ggtitle("Doctors Are Judgmental") +
  xlab("Cholesterol") +
  ylab("How Happy Your Doctor Is")
```

2.3.3 Box-and-whisker plot

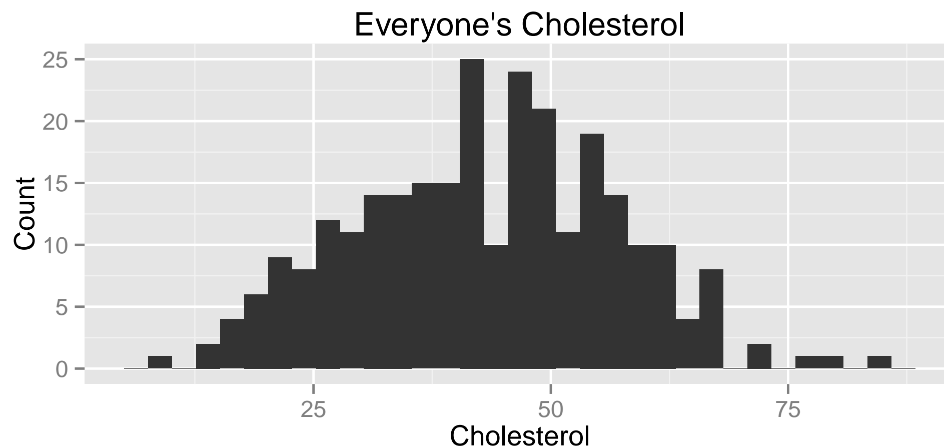
```
ggplot(d) +
  geom_boxplot(aes(x=numhamburgers, y=dochappy)) +
```

```
  ggtitle("\"You Should Cut Down on the Burgers\"") +
  xlab("# Hamburgers") +
  ylab("How Happy Your Doctor Is")
```



2.3.4 Histogram

```
ggplot(d) +
  geom_histogram(aes(x=cholesterol)) +
  ggtitle("Everyone's Cholesterol") +
  xlab("Cholesterol") +
  ylab("Count")
```

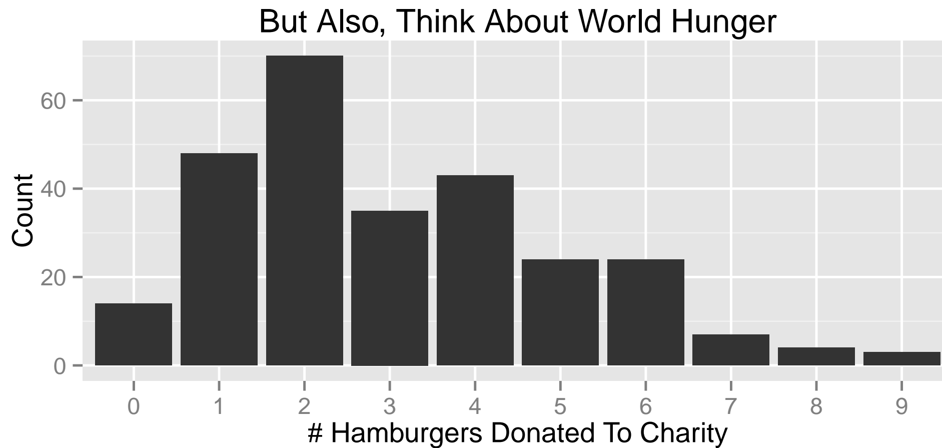


2.3.5 Bar plot

Note that the categorial variables need to be formatted as such to do a bar plot.

```
# Format the data first
d$hamcharity <- factor(d$hamcharity)
ggplot(d) +
```

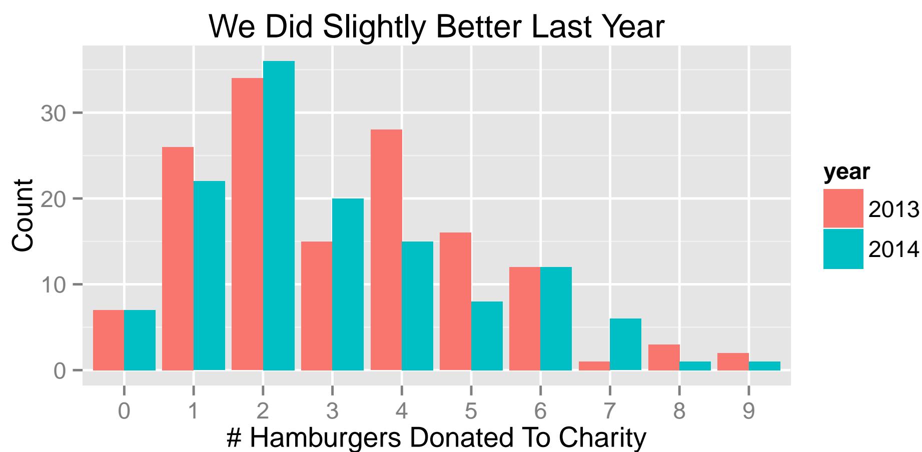
```
  geom_bar(aes(x=hamcharity)) +
  ggtitle("But Also, Think About World Hunger") +
  xlab("# Hamburgers Donated To Charity") +
  ylab("Count")
```



2.3.6 Conditional bar plot

Any conditions on which you want to split your data need to also be categorical variables.

```
# Format condition
d$year <- factor(d$year)
ggplot(d) +
  geom_bar(aes(x=hamcharity, fill=year), position="dodge") +
  ggtitle("We Did Slightly Better Last Year") +
  xlab("# Hamburgers Donated To Charity") +
  ylab("Count")
```

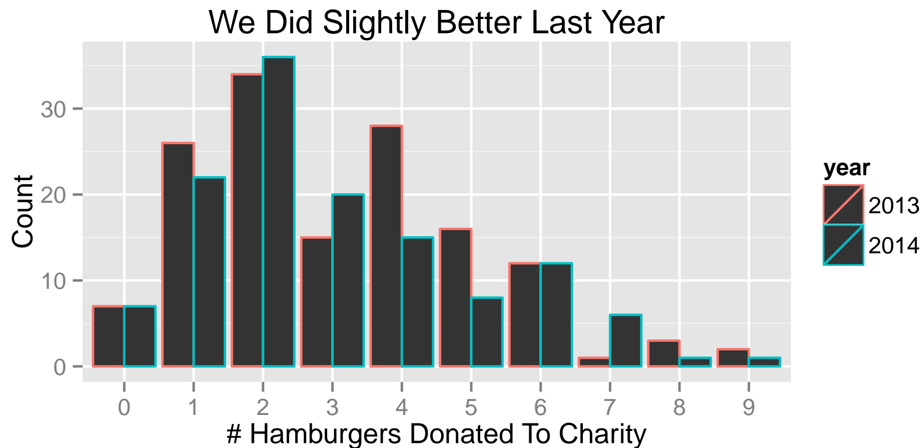


Note also that we used "fill" here and not "colour." In bar plots, "colour" refers to the outline of the bars, while "fill" actually refers to the fill color of the bars. It still looks cool if you're into designs from the 90s.[12]

[12]No judgment here. The 90s were pretty nice in a lot of ways.

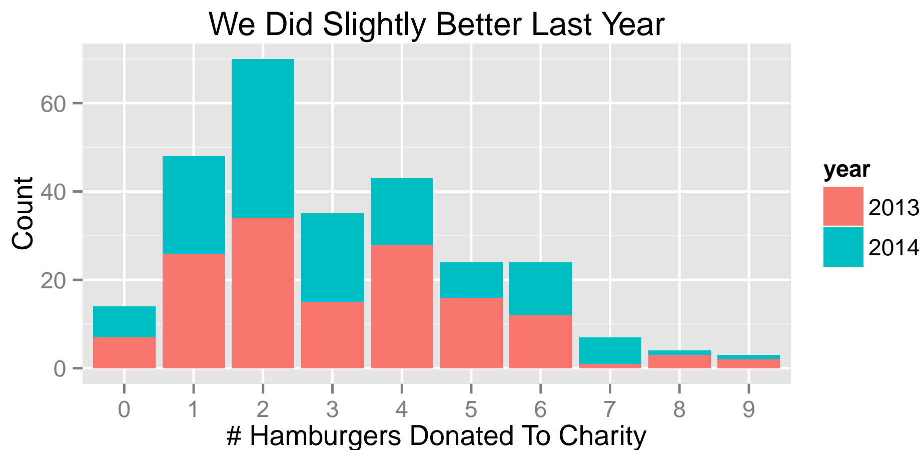
In general, solid shape colors in `ggplot()` are managed with the "fill" option, while lines are managed with the "colour" option.

```
# Format condition
d$year <- factor(d$year)
ggplot(d) +
  geom_bar(aes(x=hamcharity, colour=year), position="dodge") +
  ggtitle("We Did Slightly Better Last Year") +
  xlab("# Hamburgers Donated To Charity") +
  ylab("Count")
```



The "position" option told it to graph the categories as separate bars. Without the option, you have a stacked bar plot.

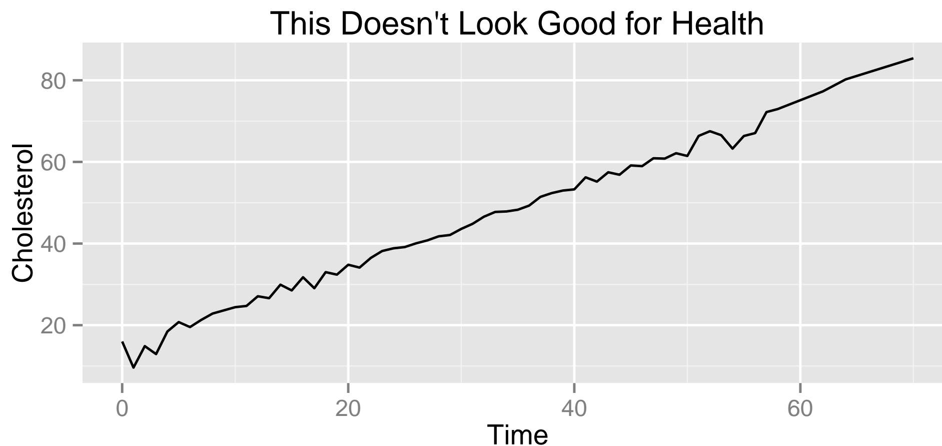
```
# Format condition
d$year <- factor(d$year)
ggplot(d) +
  geom_bar(aes(x=hamcharity, fill=year)) +
  ggtitle("We Did Slightly Better Last Year") +
  xlab("# Hamburgers Donated To Charity") +
  ylab("Count")
```



2.3.7 Line plot

For a line plot, you will want your data singularized at the x-variable level, where x values are not repeated. I have already done that for this example.

```
ggplot(dnew) +
  geom_line(aes(x=time, y=cholesterol.mean)) +
  ggtitle("This Doesn't Look Good for Health") +
  xlab("Time") +
  ylab("Cholesterol")
```



2.3.8 Re-using graph formats

This isn't so much a graph type as it is a useful tip. Sometimes, you'll find yourself using a certain set of graph format settings over and over again, and you'll get tired of re-typing those settings again and again. To make things easier, you can just store those settings in a shortcut and use that name instead.

For example, I tend to use the following a lot.

```
theme(axis.text.x = element_text(colour = "black"),
      axis.text.y = element_text(colour = "black"),
      panel.grid.major = element_blank(),
      panel.background = element_blank(),
      axis.line = element_line(colour = "black"))
```

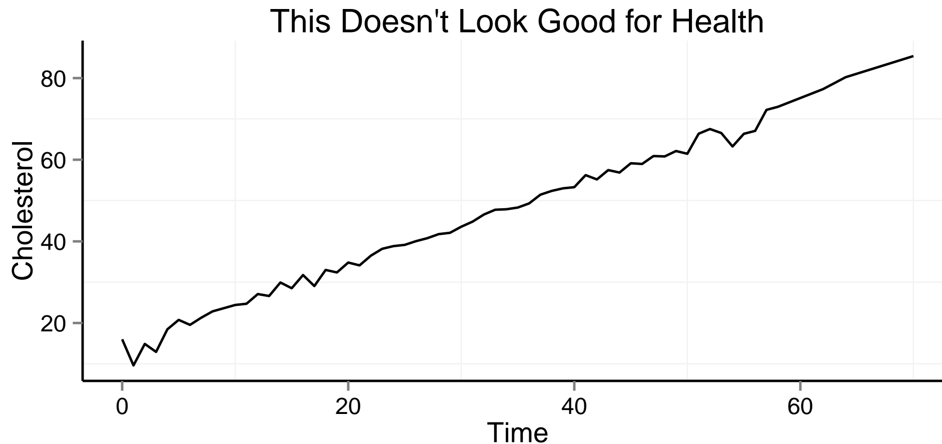
Those code makes the axis labels black, takes away the background grid, removes some grid lines, and adds black lines for the axes. I can do the following to store this setting in a container called "myformat."

```
myformat <- theme(axis.text.x = element_text(colour = "black"),
      axis.text.y = element_text(colour = "black"),
      panel.grid.major = element_blank(),
      panel.background = element_blank(),
      axis.line = element_line(colour = "black"))
```

Now, I can use it whenever I want, without typing out that mess above.

```
ggplot(dnew) +
  geom_line(aes(x=time, y=cholesterol.mean)) +
```

```
  ggtitle("This Doesn't Look Good for Health") +
  xlab("Time") +
  ylab("Cholesterol") +
  myformat
```



2.4 Output the data to a file

Okay, so you can see a pretty graph, but you don't know how to output it to a file. There are two ways to do it. The first, easiest way to do it is just to hit the "Export" button in RStudio and do it that way.

The second way, for those of us that are more code-inclined, is to code it. It's easy - open a graphics device at the beginning of the graph, and close it at the end of the graph. Sounds foreign? Type this.

```
png("linegraph.png", width=8, height=5, units="in", res=240)
ggplot(dnew) +
  geom_line(aes(x=time, y=cholesterol.mean)) +
  ggtitle("This Doesn't Look Good for Health") +
  xlab("Time") +
  ylab("Cholesterol") +
  myformat
dev.off()
```

```
null device
          1
```

We added two lines, the `png()` command and the `dev.off()` command, at the beginning and end of the code from the last example. The `png()` command tells R what format the output file should be, what it is called, the size, and the resolution of the image. There are analagous `pdf()`, `jpeg()`, `tiff()`, and other commands if you would like those image formats instead.

The `dev.off()` code at the bottom simply closes the graphics device. Don't worry too much about what this means. Just do it.

You may get an output message like what we got above. Ignore it. Note that you will *not* see the output of the graph in the bottom-right console when you use a graphics device to output the file.

And that's it. You've done it. Good job. Go get yourself a pizza.